THE TREE MACHINE OPERATING SYSTEM

Pey-yun Peggy Li
California Institute of Technology
Pasadena, California 91125

TM #4618

CONTENTS

I. Introduction

The tree machine has been well defined as a concurrent computing system by Sally Browning & Carver Mead [1]. Many algorithms have been developed and shown to have better time-performance than conventional sequential algorithms. And since a tree machine is being built and is expected to become operational within a year, a real tree machine (not just a mathematical model or some pieces of hardware) is about to face the world. This report is treating the bridging between the mathematical model and the hardware machine to make the Tree Machine become real !

In the following sections, I will describe the hardware structure of the tree machine processors, the software structure of the machine operating system, the detailed description of the assembler, the loader and the simulator. Some of the tree machine algorithms have been tested under this software system. Some performance evaluation is presented in the last section.

II. Hardware Structure

The Tree Machine is a tree structure multiprocessor system. It is a collection of small processors connected together as a binary tree. The processors are identical and each has its own local memory. Only local communication is allowed in the tree machine and only the root processor can communicate with the outside world. The tree machine resides in a host computer as a slave machine. The host computer runs most of the system routines of the tree machine and serves as the interface between users and the tree machine.

The tree machine processor is a conventional sequential processor. The processor modeled in my simulator is based on the instruction set and the architecture of the Caltech Tree Machine Processor (TMP). It is a 12-bit microprocessor with 4K nibble (4-bit) memory. The processor is composed of a 12-bit data path, 16 general purpose registers, program counter, memory address register(all 12 bits), instruction register(4 bits) and four Input/Output ports. Each I/O port consists of two 16-bit shift registers (one for input, one for output) and two flags. One processor can be connected with four other processors simultaneously. In the tree

machine, only three ports are used to connect the father processor and the two children processors.

The instruction set is in the variable length form. The op-code is one nibble in length. Some of the op-codes have sub-ops which are also one nibble long. Since there is no accumulator, all the instructions are register- to-register instructions. The basic addressing mode is indirect addressing (memory manipulation instructions and jump instruction). All the branch instructions are in relative addressing mode. The instruction set can be divided into memory manipulation, unary operations, binary operations, sequence control, input/output instructions and flag manipulation instructions. The length of the instructions varies from 2 nibbles to 6 nibbles. The list of the instruction set is appended in APPENDIX A.

The I/O port is designed to have two 16-bit shift registers and two one-bit flags associated with them. The message is transmitted serially between the processors. A message is composed of a 4-bit message number and one 12-bit argument. Hence, there is a restriction on at most 16 different messages in one processor. It is not necessary to have only one argument for one message. But, the arguments have to be sent in/out one at a time.

III. Software Structure

This section focuses on the construction of the interface between users and the physical tree machine. The host computer serves an interface role in the tree machine software system. Most of the system programs will run on the host computer.

The tree machine software system is illustrated in Fig.3-1. The host computer accepts programs written in some concurrent notation as input to a compiler. The compiler generates an intermediate form wich is a tree machine assembly language. The tree machine assembly language is designed to have features such that it can be either the output of the compiler or a language that users can write programs in easily. The compiler does not deal with the mapping problem from a logical tree to a binary tree. The mapping problem is left to the assembler. The assembler generates the machine code and an information file which provides the necessary addressing information for the loader. The loader will take the machine

code and load them down to the tree. After the loading is finished, the host computer will initiate the tree machine and the tree machine is ready to run. Then, the user can send messages to the host computer and the host computer will put them to the input port of the root processor. The host computer will also sense the output flag of the root processor. If there is an output message at the output port of the root, the host computer will take it out and send it back to the user.

Because there is no well defined concurrent notation which can program the tree machine properly, the compiler part has been omitted at this preliminary stage. The construction of the assembly language is based on both Sally's Notation [1] and Marina's HARMOS [4]. The definition of the connection plan in the assembly language is derived from Sally's notation. The entry procedure structure in the program definition part is the basic idea of Marina's notation. Hence, any program written in either notation can be transfered to the tree machine assembly language quickly.

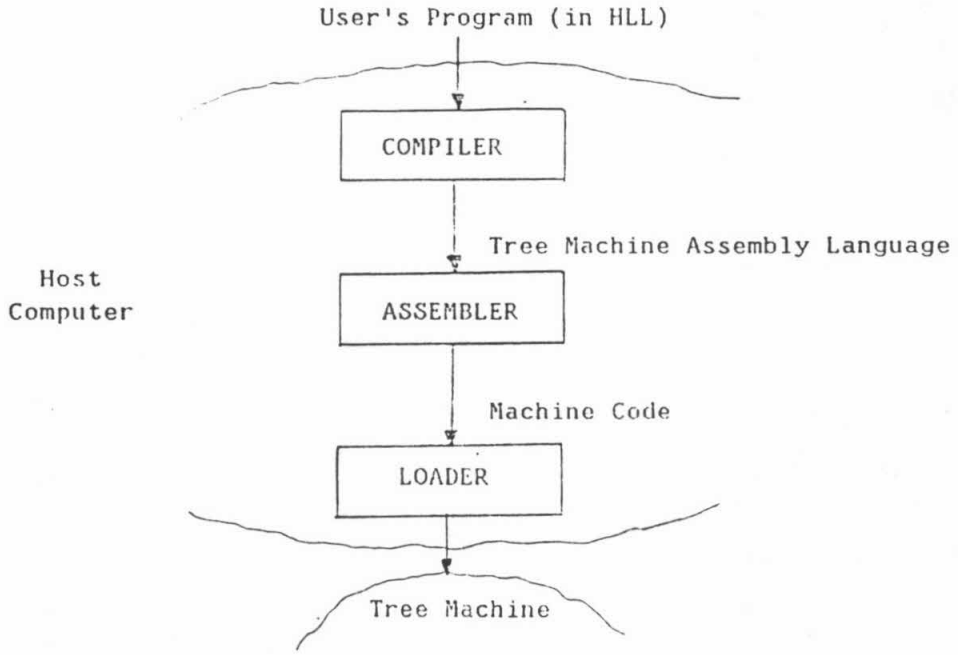


Fig.3-1 The Structure of the Tree Machine Software

IV. The Assembly Language and the Assembler

An assembly language has been designed for the tree machine. The assembly language has two major parts: the definition of the connection plan and the definition of the program code. The mnemonic instructions are based on the instruction set of the tree machine processor. The pseudo instructions define the connection plan, the ports and the memory allocation. The pseudo-macro instructions give users the capability of defining macros and expanding macros. The structure of the assembly language makes it possible to write programs that define any kind of logical tree. In other words, this assembly language has features of a high level language while it has the syntax of a low level language.

A program written in the assembly language has the form shown in Fig. 4-1. Notice that a MODULE of program is the program for one node. Some nodes in the tree may be programed in the same way. All those nodes have the same module name and store the same program code in the memory.

```
PROGRAM      progname
CONNECT
ROOT     rootName
NODE     nodenames,..
...
(the definition of the connection plan)
ENDC

MODULE     modulename
PORT      (declaration of the port name)
EXT       portname    (connect to father node)
INT       portnames,.... (connect to children nodes)

PROC   procedurename
(the code in this procedure)
....
ENDP
....
(the program code of this module)
....

ENDM      (end of module)

MODULE     modulename
(beginning of the program codes of the second module)
....
....
ENDM

....
....
END       (end of program)
```

Fig. 4-1 The Format of A Tree Assembly Program

All the instructions that appeared in Fig.4-1 are pseudo instructions. Those pseudo instructions build the skeleton of the whole program. In the following, the pseudo instructions will be described in detail.

4.1 The Pseudo Instructions

Definition of connection plan

CONNECT the head of the connection plan

ROOT define the root of the tree and the fanout of the root node. The syntax is

ROOT <identifier>(<integer>)

where <identifier> is the name of the root module,
<integer> is the fanout

NODE define all the nodes in one level of the tree. The NODE instructions following the ROOT instruction define each level of the tree from root to the leaves sequentially. The syntax is

NODE [<integer>]<identifer>[(<integer>)]
{,[<integer>]<identifer>[(<integer>)]}

If there is only one argument without a number before it, that means all the nodes in the current level of the tree have the same name, i.e. the name of that argument. For example,

NODE fool

means all the nodes in this level have the name "fool", no matter how many nodes this level has.

If there are more than two arguments after NODE, then the nodes in the current level are defined from left to right by the arguments listed in the NODE instruction. The argument with a number before it means that it will occur <integer> times repeatedly. Otherwise, it occurs just once. For instance, *NODE 3a,b,4c* means that there are 8 nodes in this level, they are a,a,a,b,c,c,c,c from left to right.

The parentheses after <identifier> define the fanout of that node. If it is omitted, that means it is a leaf node with no fanout.

SUBTREE define all the nodes in a subtree. It takes one argument which is the name of the node in the top of the subtree. The syntax is

SUBTREE <identifer>

The SUBTREE definition will be terminated by the instruction ENDS. The instructions between SUBTREE and ENDS define the nodes in the subtree level by level. The subtree definition can be nested defined.

ENDS the end of the subtree, takes no arguments.

ENDC the end of the connection plan

Port Declaration

PORT the head of the port declaration, takes no arguments.

EXT define the name of the external port which is connected to the father node. The syntax is

EXT <identifier>

INT define the names of the internal ports which are connected to the children nodes. The number of arguments is decided by the fanout of that node. Array variables can be accepted in this instruction. The syntax is:

INT <identifier> [{,<identifier>}|(<integer>:<integer>)]

For example, INT 1,r means that the node has two internal ports named '1' and 'r'. *INT c(1:3)* means it has three internal ports named c(1), c(2) and c(3).

Memory Allocation

BWS define a block of words (12 bits or 3 nibbles) in the memory. It takes one integer argument which is the number of words in this block. The syntax is

<label> BWS <expression>

The legal format of <expression> is

<operand>[<operand>][(+|-)<operand>]*

<operand> can be an integer, a variable name or a '*' (the current address of the location counter). For example, A+2, 3*B, C*A-3, *-1 are legal expressions and 2+A*2, A*2+B*3 are not.

DWF allocate a word field in the memory. It takes one argument which is the initial value stored in this memory location. The syntax is

<label> DWF <expression>

EQU assign a value to a constant literal. It takes one argument which is the value of that literal. The syntax is

<label> EQU <expression>

Program Heading and Ending

PROGRAM the head of the program. It takes one argument as the program name. The syntax is

PROGRAM <identifier>

END the end of the program. It is always the last statement in a program.

MODULE the head of the definition of one module. It takes one argument as the module name. The syntax is

MODULE <identifier>

ENDM the end of the program of one module.

PROC the head of one entry procedure. It takes one argument as the entry name of that procedure. The syntax is

PROC <identifier>

ENDP the end of one procedure

Macro Instruction

MACRO the head of a macro definition. Macro definition can be nested. Macro calls is also allowed in a macro definition.

EMC the end of the macro definition

Sequence Control

DO the head of a do loop. It takes one argument which is the number of iterations. The do loop will end with an "EDO" statement. All the instructions between "DO" and "EDO" will be repeated n times in the object code, where n is the argument value after "DO". The syntax is

DO *⟨expression⟩*

DO statements can be nested.

EDO the end of a do loop

4.2 Macro Definition and Macro Call

The tree assembly language has the capability of defining macros or expanding macros. A macro definition starts with a pseudo-op MACRO and ends up with a pseudo-op EMC. An example of a macro definition is shown in Fig.4.2. The second line in the macro is the macro instruction name. In the operand part of this line, it specifies the dummy arguments of the macro. All the arguments should be preceded with a '#' character to be distinguished from an assembly language symbol. Following the name line is the sequence of instructions being abbreviated by the macro name.

```
        MACRO                              !beginning of a macro;
        INCR  #ARG1,#ARG2,#ARG3,#LAB       !Macro name;
L#LAB   ADD   1,#ARG1                      !codes;
        ADD   2,#ARG2
        ADD   3,#ARG3
        EMC                                !termination of the macro;
```

Fig. 4-2 A macro definition

Once a macro has been defined, we can use the macro name as an operation mnemonic in an assembly program. The assembler will expand the corresponding instruction sequence into the source program when a macro call occurs. The dummy arguments in a macro definition will be replaced by the actual parameters in a macro call instruction. Fig. 4-3 shows a piece of source program with a macro call INCR and the expanded code.

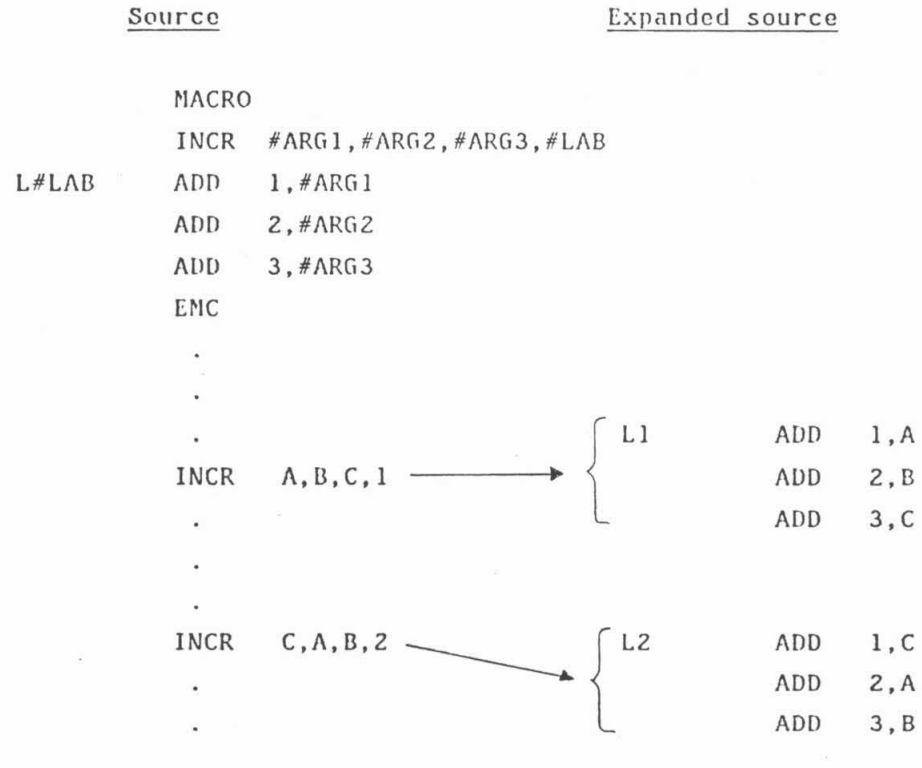


Fig. 4-3 Macro Expansion

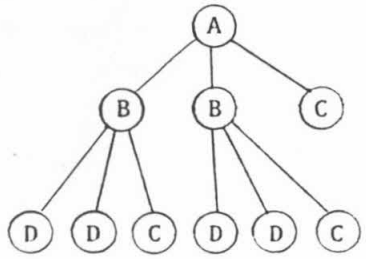
A macro call or another macro definition can be included within a macro definition. It is possible for a macro to call itself, as long as

this does not cause an infinite loop. (It can be achieved by inserting the suitable DO and EQU statements). The macro instruction processor is implemented in pass 1 of the assembler. The data structure and the algorithm of the macro processor will be described in Section 4.6.

4.3 The Representation of the Connection Plan

To write a program in Tree Assembly Language, it is necessary to represent the connection of the logical tree by the pseudo instructions. Three examples will show how it works.

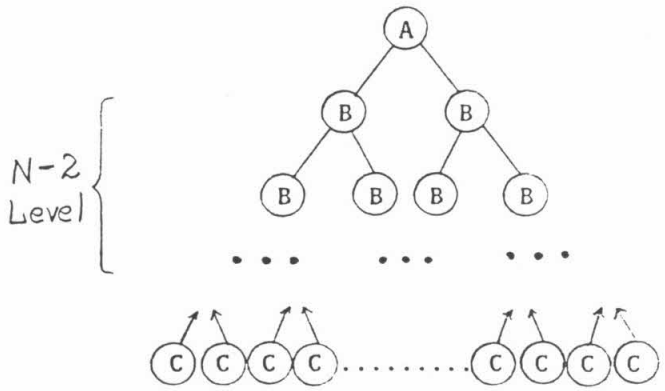
Example 1 A two-level 3-ary unbalanced logical tree with four different modules A,B,C and D



The code in assembly language looks like:

```
CONNECT
   ROOT     A(3)
   NODE     2B(3),C
   SUBTREE  B
      NODE     2D,C
   ENDS
ENDC
```

Example 2 A N-level binary tree with three different modules A,B and C

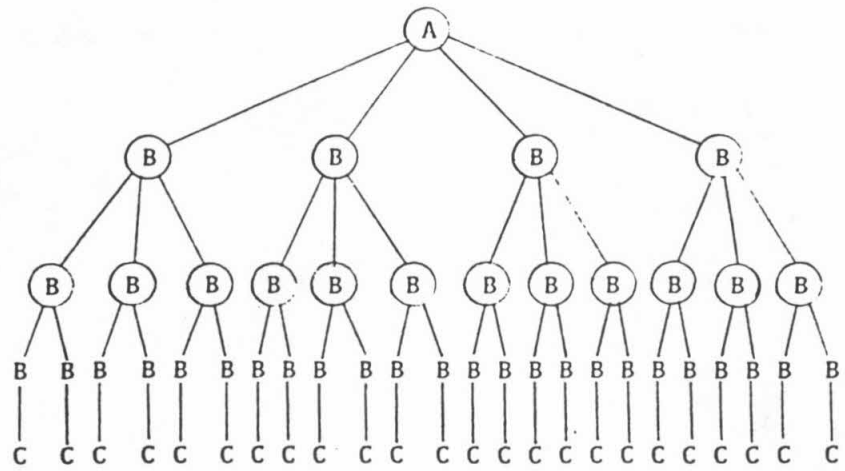


The code in assembly language looks like:

```
CONNECT
   ROOT     A(2)
   DO       N-2
     NODE     B(2)
   EDO
   NODE     C
ENDC
```

Example 3 A N-level logical tree with different fanout at each level, the root has fanout N-1, the nodes at level 2 have fanout N-2, The fanout decreases level by level. The second level from the bottom has fanout 1 only.

(Example of Sally's Travelling Salesman Tree)



(N = 5)

The code in assembly language looks like:

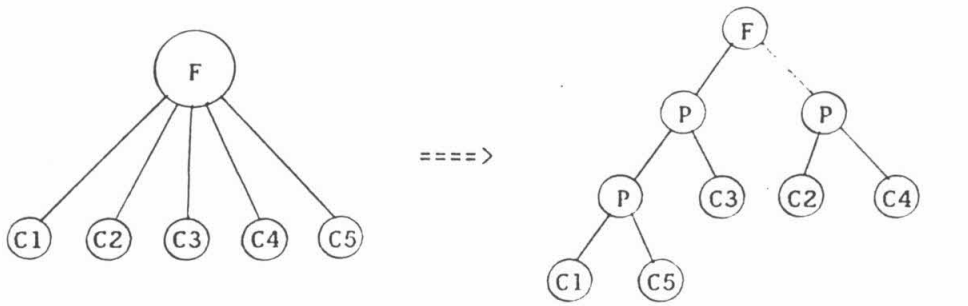
```
          CONNECT
             ROOT   A(N-1)
K            EQU    N-2
             DO     N-2
               NODE   B(K)
K              EQU    K-1
             EDO
             NODE   C
          ENDC
```

From the three examples, we can find that by using the DO statement and EQU statemet, any kind of tree structure can be represented easily and neatly in the assembly language.

4.4 The Padding Problem

The padding problem is encountered when an n-ary logical tree is to be mapped to a binary tree. A padding processor is a processor which is inserted between a father node and the children nodes in the logical tree in order to establish the necessary fanout. A padding processor acts as a communication node in the binary tree.

There are two ways to do the padding. The first one is proposed by Sally Browning [1,3]. In Fig. 4-4, half of the children nodes are mapped to the left subtree of the father node and half to the right. The child node with the odd number in the logical tree is assigned to the left of the father and the child with even number is assigned to the right. Hence, the binary tree doesn't preserve the order of the children in the logical tree. The second way to implement padding has the same structure as the first one. The difference is that the sequence of the n-ary tree is preserved in the binary tree as shown in Fig. 4-5.



(P: padding processor)

Fig. 4-4 Method 1 for padding

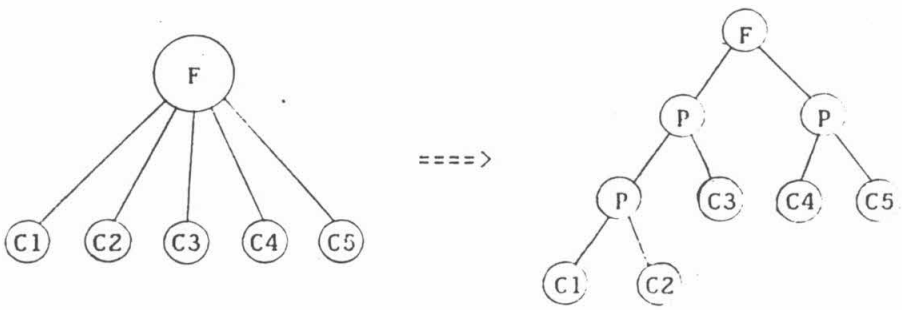


Fig. 4-5 Method 2 for padding

When the father node wants to send a message to one of the children, it is first necessary to decide if that specific child is in the left or the right branch. Secondly, it is necessary to decide if there is any padding processors between the father node and the child. If there is/are some padding processor(s), the father node has to provide the padding processors with proper information so that the padding processor can select

the right path.

Associated with each output statement in the father node, there exist in the padding processors a corresponding entry procedure with the same name as the output message. Two arguments will be received from the father node. One describes the depth of the subtree under this padding node or the total number of children nodes under this padding node. The other argument indicates the number (identity) of the destination child.

When the node is going to receive an input message from one of its children, steps similar to the steps described above has to be followed to find the communication path to the child. Hence, an extra message to the padding processor in that path is necessary. A corresponding entry procedure INPUT in the padding processors receives the message from the father and selects the desired child, asks for the message from that child and then sends it to the father node.

Fig.4-6 shows the converted code for input/output statements of the father processor and the code of the input and output procedures in the padding processors for Method 1 shown in Fig.4-4. Fig.4-7 is the code for Method 2. The code is written in Marina's Notation [4]. In the form *C(i).out?X* and *C(i).control!X*, C(i) is the name of the child processor. out and control are the name of the ports. control is the only port which can get an output message from the father node. ! means output and ? means input. X is the message name. Message X has no argument in this example.

Father Processors: (n is the fanout)

Case 1: C(i).out?X

```
Convert to:
     [odd(i) --> 1.control!input((n-3)/2,(i+1)/2); 1.out?X
     |even(i) -->  [n=3 --> skip
                   |n>3 --> r.control!input((n-4)/2,i/2)
                   ]; r.out?X
     ]
```

Case 2: C(i).control!X

```
Convert to:
     [odd(i) --> 1.control!X((n-3)/2,(i+1)/2)
     |even(i) -->  [n=3 --> r.control!X
                   |n>3 --> r.control!X((n-4)/2,i/2)
                   ]
     ]
```

Padding Processors:

```
Procedure input(d,p);
  d,p: integer;
  in?input(d,p);
  [d=0 and odd(p) --> 1.out?X
  |d<=1 and even(p) --> r.out?X
  |d>0 and odd(p) --> 1.control!input((d-1)/2,(p+1)/2);
                      1.out?X
  |d>1 and even(p) --> r.control!input((d-2)/2,p/2);
                     r.out?X
  ]; out!X
EndProcedure

Procedure X(d,p);
  d,p: integer;
  in?X(d,p);
  [d=0 and odd(p) --> 1.control!X
  |d<=1 and even(p) --> r.control!X
  |d>0 and odd(p) --> 1.control!X((d-1)/2,(p+1)/2)
  |d>1 and even(p) --> r.control!X((d-2)/2,p/2)
  ]
EndProcedure
```

Fig.4-6 Padding Codes with even-odd separated order

Father Processor:

Case 1. C(i).Out?X

Convert to:

```
[ i <= (n+1)/2 --> 1.control!input((n+1)/2,i); 1.out?X
| i >  (n+1)/2 --> [ n=3 --> skip
                   | n>3 --> r.control!input(n/2,i-(n+1)/2)
                   ]; r.out?X
]
```

Case 2. C(i).Control!X

Convert to:

```
[ i <= (n+1)/2 --> 1.control!X((n+1)/2,i)
| i >  (n+1)/2 --> [ n=3 --> r.control!X
                   | n>3 --> r.control!X(n/2,i-(n+1)/2)
                   ]
]
```

Padding Processors:

```
Procedure input(n,i);
   n,i: integer ;
   in?input(n,i);
   [i <= (n+1)/2 --> [ n<=2 --> skip
                     | n >2 --> 1.control!input((n+1)/2,i)
                     ]; 1.out?X
   | i > (n+1)/2 --> [ n<=3 --> skip
                     | n >3 --> r.control!input(n/2,i-(n+1)/2)
                     ]; r.out?X
   ];  out!X
Endprocedure

Procedure X(n,i);
   n,i: integer ;
   in?X(n,j);
   [i <= (n+1)/2 --> [ n<=2 --> 1.control!X
                     | n >2 --> 1.control!X((n+1)/2,i)
                     ]
   | i > (n+1)/2 --> [ n<=3 --> r.control!X
                     | n >3 --> r.control!X(n/2,i-(n+1)/2)
                     ];
   ]
Endprocedure
```

Fig.4-7 Padding Codes with children in order

The generation of padding codes is performed in the first pass of the assembler. The code in Fig.4-7 is rewritten into four Macro's in the assembly language. The assembler treats the padding generation as ordinary macro exapansion. The details will be described in Section 4.6.

Table 4.1 is a simple comparison between an n-level, m-ary logical tree and its corresponding binary tree with padding processors.

Table 4.1
A Logical Tree vs. a Physical Tree

	Fanout	Level	# of Nodes Used	Total Nodes
Logical	M	N	$(M^{N+1}-1)/(M-1)$	$(M^{N+1}-1)/(M-1)$
Physical	2	$N \times \log_2 M$	$1+2(M^N-1)$	$2^{N \times \lceil \log_2 M \rceil + 1}-1$

4.5 The Three-Pass Assembler

The three-pass assembler combines the macro processor (pass 1) and the assembler (pass 2 & 3) together. Except for generating the machine codes, it also transforms the logical tree to a binary tree and provides the necessary information for the loader and the simulator.

The major functions of the assembler are as follows:

1. Generate the machine code
2. Expand system macros (I/O macros, dispatch macros) and user defined macros
3. Insert the code of padding processors when the logical tree is not binary
4. Create a tree structure which is isomorphic to the physical tree machine. This tree structure is an important tool for generating the address information to the loader and running the mapping algorithms without touching the physical tree.
5. Do the logical-physical conversions, such as giving a number to each logical port name, matching all the message names with distinct numbers, creating a dispatch table which includes the starting address of each entry procedure, etc.
6. Generate three files. One is the program object code. Another one is a module name file derived from the isomorphic tree structure

and it is an input file to the loader. The third file contains the message name/number pairs of the root module. It serves as a user interface file which will be called in the simulator.

The structure of the assembler is shown in Fig. 4-8. It takes the source file with extension .TSM as the input, the pass 1 does the macro expansion and padding code generation and generates a temporary file (.PS1) which contains the expanded assembly program. The file .PS1 is the input to pass 2 of the assembler. In the second pass, the symbol table is constructed and the expressions and literals are evaluated. A new temporary file (.PS2) will be generated. The file .PS2 is the input to pass 3 of the assembler. The code generation is done in pass 3 and three files will be generated, that is, the object code file (.TBJ), the module name file (.TN) and the root message file (.TBL).

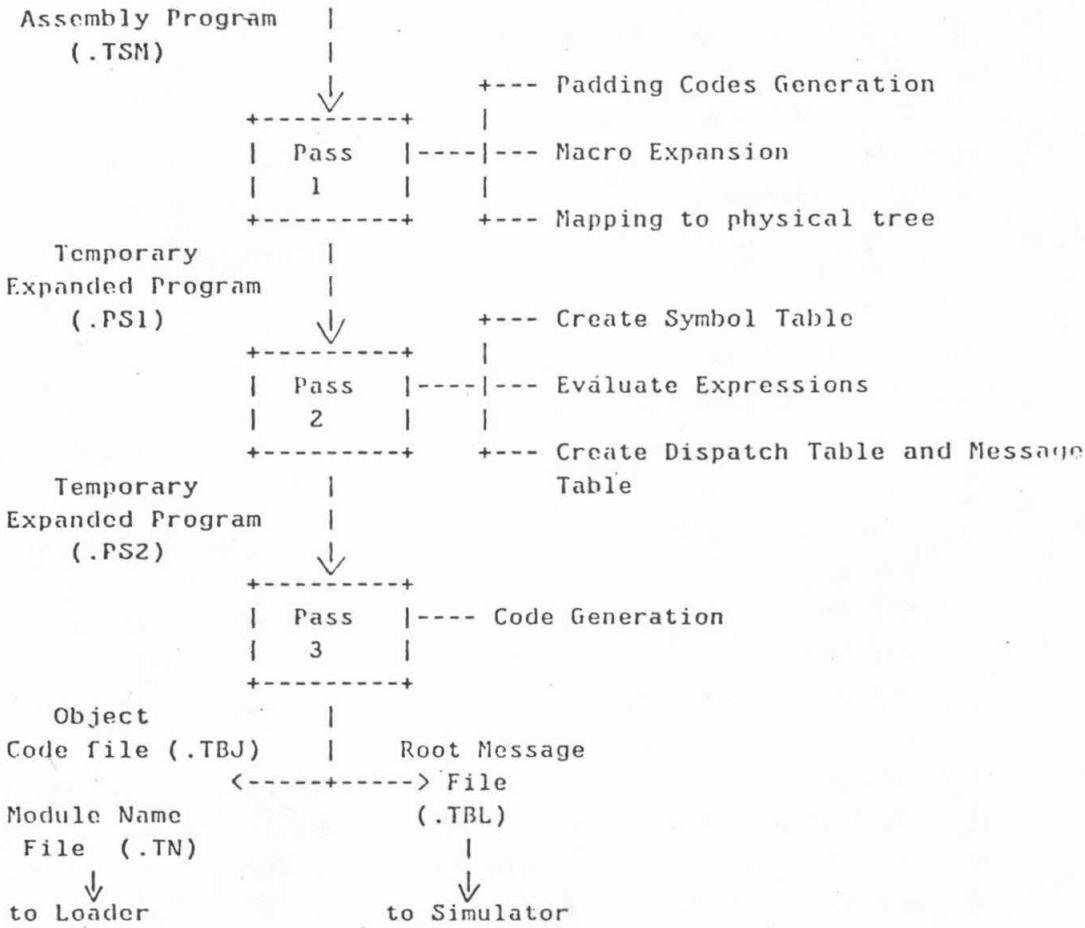


Fig.4-8 The Struction of the Assembler

4.6 Pass I: Macro and Padding Expansion

The first pass of the assembler is a macro processor. In addition to being a traditional macro processor, it also inserts the padding code and performs the do loop expansion. Besides, it creates an isomorphic binary tree structure containing the processor name in each node.

The flow chart is shown in Fig.4-9. Each block in the diagram is a subroutine call. I will describe the subroutines ReadLine and CreateTree in detail.

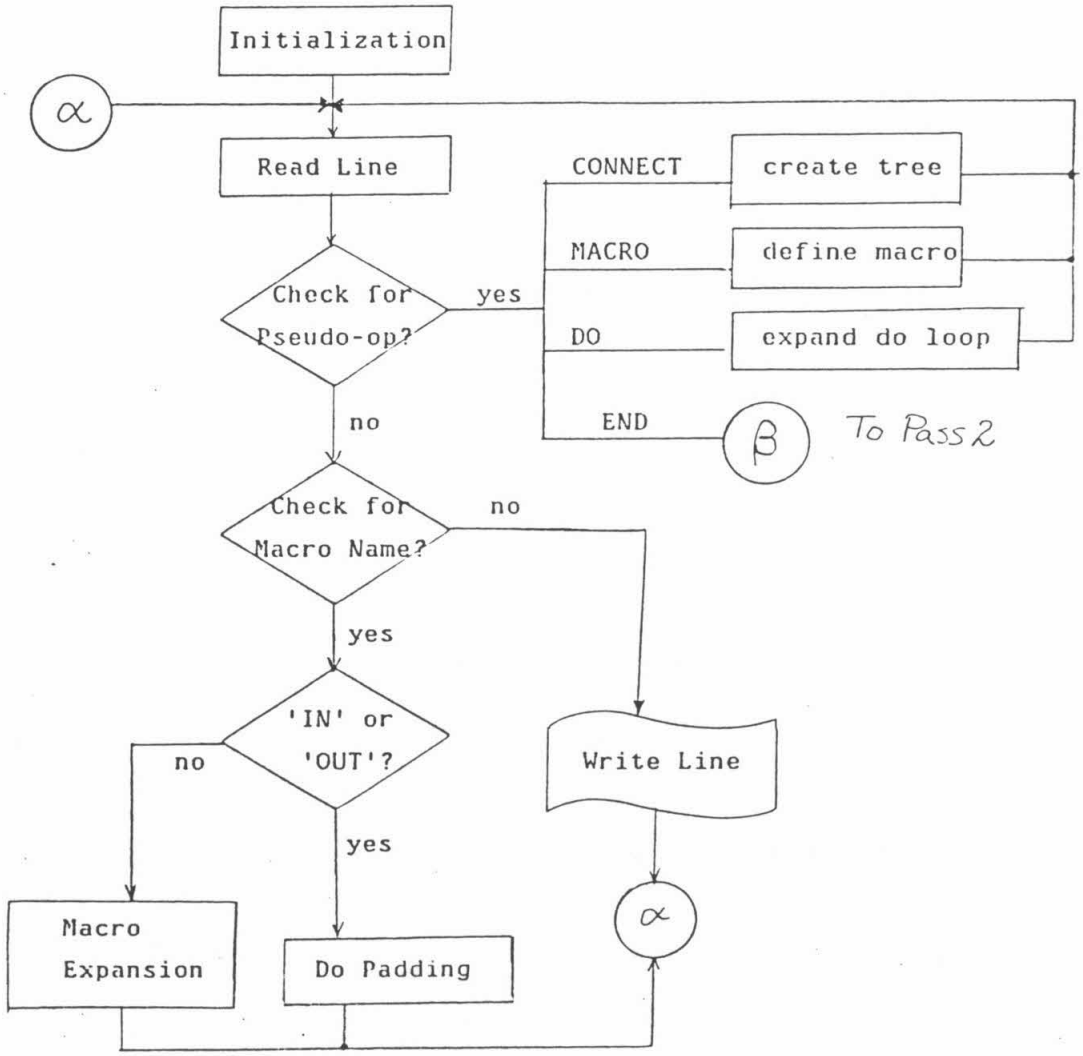


Fig. 4-9 The Flow Chart of the Pass 1

Before describing the detailed construction of the subroutines, I will list the supporting data structures for the program. Two basic table structures are widely used in this program, vector and treeDictionary. Both of them are defined as a subclass of the super class thing. vector is an array with arbitrary length. The entry of vector is a thing, e.g. an integer, a string, a boolean, a data pair or an array, etc. TreeDictionary is a tree dictionary. The entry is a key/attribute pair. All the entries are sorted by the key word. The tree dictionary has the functions insert, delete and search.

Data Structure:

Port Table --- TreeDictionary
Module Table --- TreeDictionary

For Macro Expansion:

Macro Definition Table --- vector
Macro Name Table --- TreeDictionary
Macro Stack --- vector (used for the macro call within a macro definition)
Stack Pointer --- Integer
Macro Loop Counter --- Integer (used for the nested macro define)
Argument No. --- Integer (Indicate the length of arguments for the current expanded macro)

For Padding Codes Insertion:

Padding Definition Table --- vector
Padding Pointer --- Integer

For Do Loop Expansion:

Do Loop Stack --- vector (store the instruction sequence quated by DO)
Do Stack Pointer --- Integer
Statement Length --- Integer (indicate the length of statements inside the current do loop)
Do Scope Counter --- Integer (counter for nested do loop)

Flags:

Macro In Do Flag --- A macro call occurs within a Do loop
Padding Flag --- Boolean (insert the padding codes)

fanout --- integer (indicate the fanout of the current module)
Source file
Output file

Subroutine ReadLine will read one statement from four possible sources: 1) the source file, filename.TSM; 2) the macro definition table when a macro call has been detected in the source file; 3) the padding table which stores the codes for a padding processor; 4) the do loop stack for a do loop expansion.

The supporting data structure for macro, padding and do loop expansion has been listed before.

The macro definition table stores the user defined macro or system macro statements sequentially. A macro name table associated with the macro definition table stores the macro name and a pointer pointing to the first statement of that macro in the macro definition table. When a macro call is met, a stack will be created to store the pointer to the macro definition table and the actual argument list. A stack pointer will point to the first entry of the last set of macro information in the stack. The stack pointer is initialized to -1. The previous stack pointer is always stored in the first entry of the next set of macro information. That provides a path to retrieve the outer scope macro when a macro call occurs within a macro definition. Fig.4-10 illustrates the relation among these three tables. When the stack pointer is not equal to -1, that means a macro call has been encountered and the next input line should be taken from the macro definition table. The second entry in the stack stores the address of the next input statement in the macro definition table. The value will be incremented after each reading. The Readline routine will keep reading the contents in the macro definition table until an EMC is read. When an EMC has been detected, the current set in the stack will be destroyed and the stack pointer is set to the value of the previous stackpointer which is stored in the location of the current stack pointer.

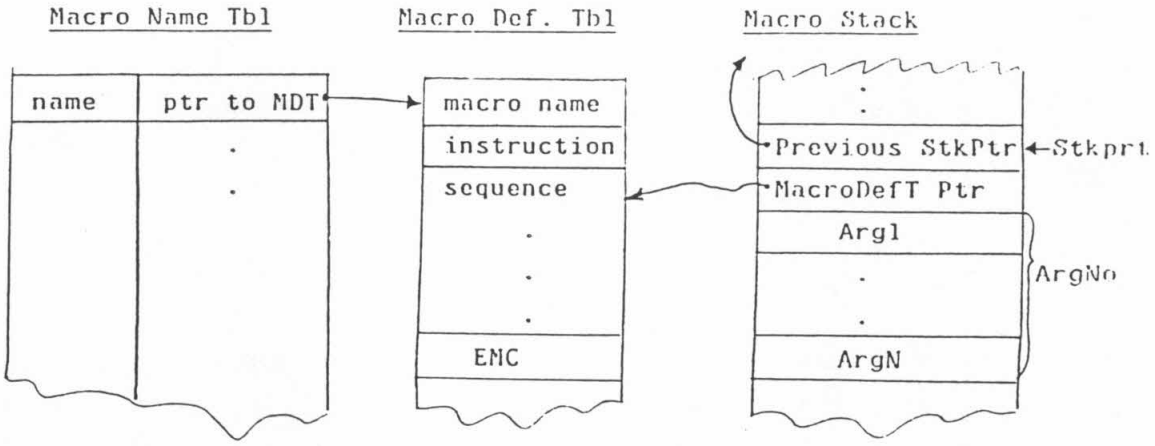


Fig.4-10 The Data Structure of Macro Expansion

For do loop expansion, a do loop stack is built to store the statements within the do loop. The structure of the do loop stack is similar to the macro stack except that the do loop stack stores the instruction sequence instead of the argument list. There is one extra entry in the do loop stack keeping the number of iterations for the do loop. The structure of the do loop stack is shown in Fig.4-11. When a DO statement is detected, the assembler creates a stack and push the do stack pointer, the number of iterations , a pointer (initialized to 3) and the source codes following DO statements into the stack. The do loop will terminate by an EDO statement, and then the do stack pointer will be set to the starting entry of this do loop in the stack. Then, the ReadLine routine starts to read the codes in the do stack. The third entry in the stack points to the next input line and will be incremented after each reading. When the ReadLine routine gets the EDO statement in the stack, it will decrease the value in the second entry of the stack by 1 and check if the value is equal to 0. If no, then reset the value of the third entry to its initial value (3) or destroy the current stack and reset the do stack pointer to its previous value.

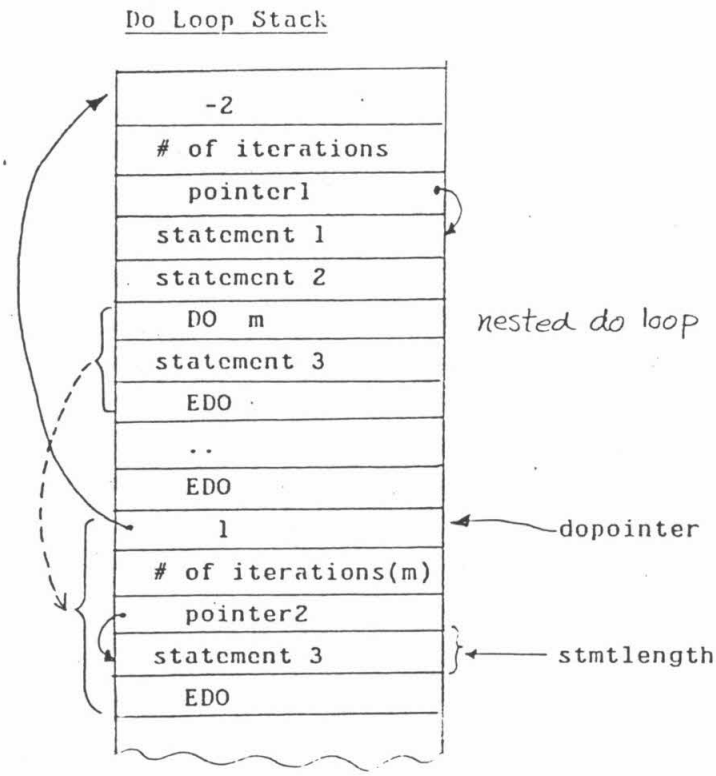
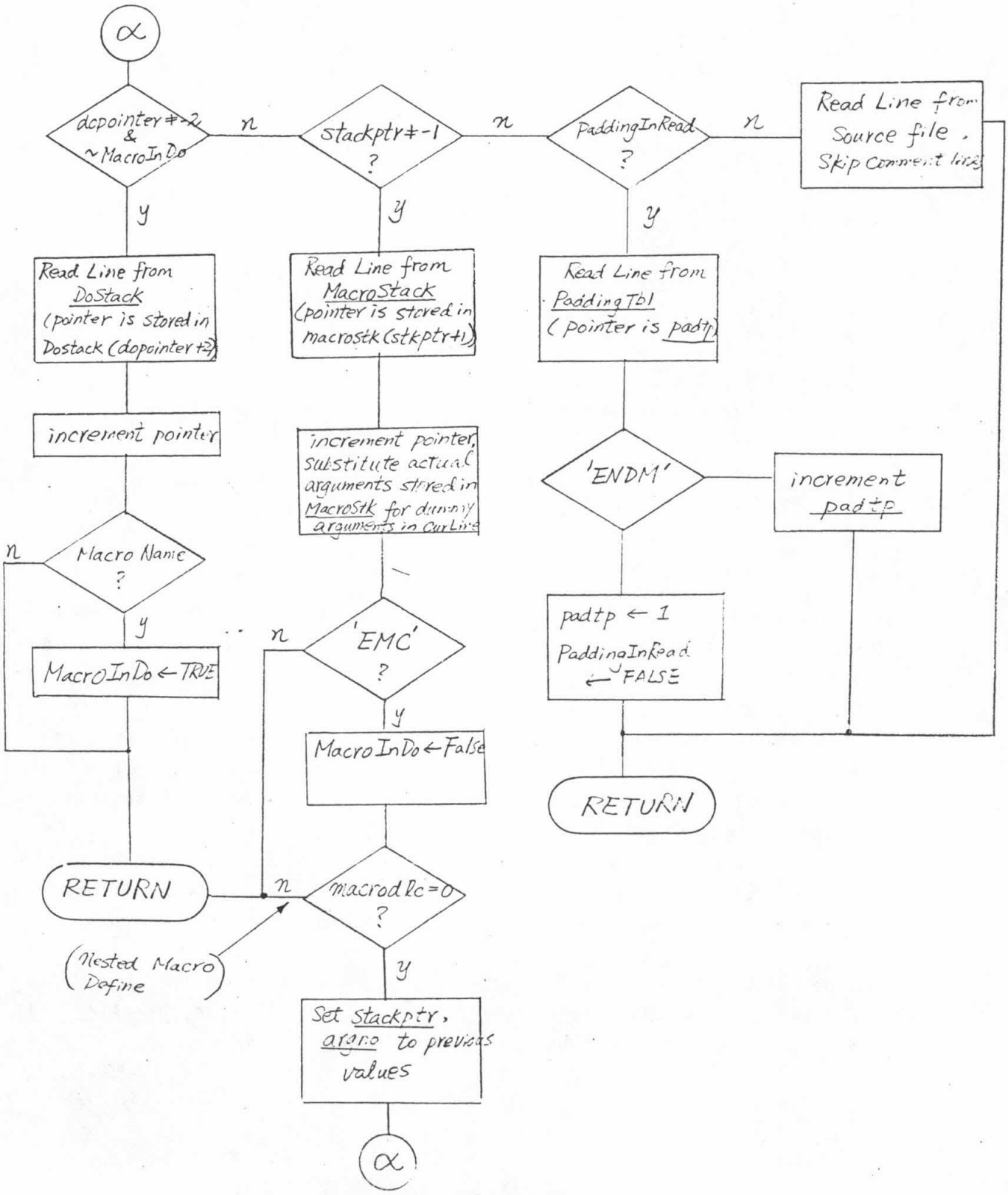


Fig.4-11 The structure of the do loop stack

When the logical tree is not binary, the padding processors should be inserted into the source code. There are four system macros reserved for the padding insertion. They are macro IN, OUT, INPUT and OUTPUT. The first two are used for the conversion of input/output statements in the father nodes. The last two define the entry procedures of the padding processors. At the dopadding subroutine (Fig.4-9), if the fanout of the current module is greater than 2 and the port name of that I/O statement is not the fathername of that module, then IN or OUT will be treated as a macro name and a macro expansion will occur. Besides, a corresponding statement for the padding processors will be generated and stored into the padding table. Notice that the head and the end of the padding processor will be generated when MODULE and ENDM instructions of the current module are detected. Once the ENDM instruction of the current module is met, the assembler sets the flag paddingInRead and then the subroutine readline will read the contents in the padding table until the ENDM is met.

The flowchart of the subroutine ReadLine is drawn in Fig.4-12.



4.7 Pass II: Table Construction and Pass III: Code Generation

Pass 2 and Pass 3 together is just like a conventional 2-pass assembler. The major difference is that we have to generate one set of object code for one module so each module must have its own data structure. CLASS module includes all the tables, variables of one module. Object current Module keeps the module currently being parsed by the assembler. It will be stored to the Module table when the parsing is finished.

The symbols defined in each module are local. But, the message names are global to some of the modules. The dispatch table is derived from the message name too. The message table and the dispatch table can not be built without some global information (more specifically, the information from its father module).

A machine-op table which is presorted by the mnemonic names contains the mnemonics, the op codes, the sub-op codes (if specified) and the length of each op. The assembler searches this table to get the next location counter or to generate the machine code.

Data Structure:

```
Machine OP Table --- presorted text array
Module Table --- vector of CLASS module's
current Module --- CLASS module (CLASS module contains all the necessary
                   informations of one module)
In CLASS module:
   myNam --- Text, module name
   myId --- Integer, module id
   Message Table --- TreeDictionary
   Dispatch Table --- TreeDictionary
   Symbol Table --- TreeDictionary
   Literal Table --- TreeDictionary
   memory --- vector of integers (store the machine codes during Pass 3)
   Location Counter --- Integer
```

4.8 Output Form of the Assembler

There are three files generated by the assembler. The first one is the object code file with the same filename as the input file of the assembler but with extension .TBJ. The form of the .TBJ file looks like the form in Fig. 4-13

```
! MODULE A
! LENGTH 210 NIBBLES
001046180000000000000000000000000000000000000000000000000B3405E115D314DC13D53DB4ED9
914BC740C7D0C7E092705C3991B93CB4DE7CB40CBC0CBE09080CB405E115D314DC13D53DB4ED9914
BC7D1C7E1B75291F05C3991393CB4DE7CBC1CBE1BB52C3526F12A17F
! END OF MODULE A
! MODULE B
! LENGTH 72 NIBBLES
003018180000000000000000000000000000000000000000000000000B340C740B752C3526F12A17F
! END OF MODULE B
! MODULE C
! LENGTH 64 NIBBLES
404016180000000000000000000000000000000000000000000000000B340C3426F12A17F00
! END OF MODULE C
FFF
```

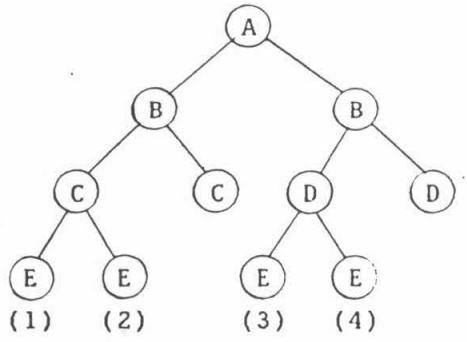
Fig.4-13 The form of .TBJ file

All the lines starting with '!' are comment lines and will be neglected by the loader. The code is hexadecimal numbers, one number will be stored into one location of the memory. There are three modules, A, B and C, in the example file. The first six numbers in each module are the header. The first three digits are the module id. In this case, module A has id 001, module B is 003 and module C is 404. Notice that the id of module C is 404, i.e. the second MSB bit of the 12-bit module id is set to 1. That indicates module C to be a leaf module in the tree. The second three digits in the header are the count of the program length in the unit of words (3 nibbles). For example, module A has $46_{16} = 70_{10}$ words(= 210 nibbles) codes and module B has $18_{18} = 24_{10}$ words (= 72 nibbles) codes. Following the header, the first 48 digits are the contents of the dispatch table, storing the starting addresses of entry procedures. Three consecutive 'F' in the last line in Fig.4-13 is the end marker of the file.

The second file generated by the assembler is the module name file with extension .TN. It contains the module id's of all the nodes in the tree. Each module id is represented by a 3 digit hexadecimal number. The

order of the module id's is from top (the root) to bottom, left-rigth alternatively. The file ends up with a three digit end marker "FFF". For example, the module name file of the tree in Fig.4-14 will look like the form shown as follows:

001002002003004003004405405000000405405000000FFF



module id
A:001
B:002
C:003
D:004
E:405

Fig.4-14 An example Tree with five different modules

In other words, the node sequence in the .TN file is A, B(left), B(right), C(the leftmost), D(the left son of the second B), C(the second), D(the rightmost), E(no. 1), E(no. 3), "000" (empty node in the left branch of the unbalanced tree), "000"(empty node in the right branch), E(no. 2), E(no. 4) and two more "000". The .TN file is used to load the module id's into the tree machine during downloading. The detailed downloading algorithm will be described in section 5.1.

The third file generated by the assembler is the root message file with extension .TBL. In the .TBL file, the message name/number pairs of the root processor are listed. The simulator will use it to get the logical name of each message number. Hence, the .TBL file is just a interface file for the simulator use.

V. Down Loading and the Bootstrap Loader

After the machine codes have been generated, the host computer has to down load the codes into the tree machine. There may be several different modules in a tree. The program has to be loaded into the right nodes in the right memory location. A bootstrap loader which is preloaded into all

the nodes in the tree will select the right codes for each node and store them into memory.

5.1 The Loading Algorithm and Its Performance

The loading algorithm is designed under the principle of loading the program code once only. No matter how many nodes share the same code or how random those nodes distribute, all the nodes containing the same module will be loaded simultaneously. Under this principle, the loading time depends only on the total amount of machine code, but is independent of the size of the tree.

How can we load the program code simultaneously into all the equivalent nodes? Remember that during pass 1 of the assembler, we build a tree structure which has the same structure as the tree machine and it contains the module name and a one-one mapped 12-bit module id in each node. The assembler generates a module name file which contains all the module id's in the tree (reference to section 4.8). The module name file is passed into the tree from the root. The root stores the first module id into one of its registers and passes the other ids down to its left and right sons alternatively. All the other nodes in the tree do the same work as the root, i.e. takes the first id as its own name and passes the others down to its subtrees until the end marker "FFF" is met. When the end marker has been detected, the node is ready to read the program code. In the object code file, the module id and an integer is located at the beginning of each module's code. The integer specifies the length of the module's code. The node in the tree will first receive the module id and compare it with its own id. If they are matched, then the integer is read, and the proper number of words are read in and stored into the memory. If the node isn't a leaf node (the second MSB of the module id of a leaf node will be set), then it also passes all the code it receives down to its left and right descendents. In this way, the program code will be loaded into all the nodes in the tree without passing it down several times or without a complex addressing scheme.

The performance of the loader is determined by two factors: one is the loading time of the module name file, the other is the loading time of the program code. The loading time of module ids depends on the size of the tree. The loading time of the program code depends on the total amount of program code but is independent of the size of the tree. Since one module

id only occupies 3 nibbles, the module name file is comparatively short when a tree is not too big. The time of loading the program code is the dominant factor for the loader performance. But, when the tree grows, the total number of nodes in the tree increases exponentially and the time for loading the module ids will also grow exponentially. The time for loading the program code remains unchanged as long as the number of different modules does not increase. In that case, the loader performance will be determined by the loading time of module names.

5.2 The Bootstrap Loader

The bootstrap loader is written in the assembly language according to the algorithm described above. The length of the bootstrap loader is 300 nibbles. The processors in the tree are first initiated to the loading mode. Then the bootstrap loader is loaded into all the nodes in the tree. After the bootstrap loader has been loaded into the tree, the processors are set to the execution mode and the program counters are set to the beginning address of the bootstrap loader. Since the dispatch macro is the same for all the nodes, the code of the dispatch macro will be appended to the end of the bootstrap loader and be loaded into the tree in the loading mode.

The allocation of the memory of each node is shown in Fig.5-1

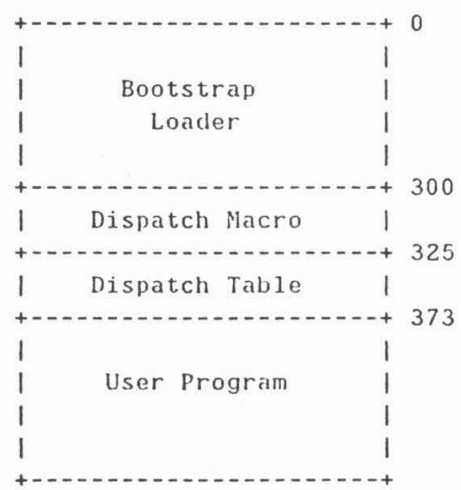


Fig.5-1 The Memory Allocation of the Tree Machine Processor

The average downloading time summary in Table 7.1 can be formulated by the following equation

Downloading time = 3 x (total nibbles of program code + 3 x total number of nodes)

VI. The Simulator

The tree machine simulator is written in Simula and running on DEC-20. It is a functional simulator at ISP level. Two major CLASSes are the heart of the simulator. CLASS processor simulates one processor of the tree. CLASS tree is a recursive tree containing CLASS processor as the node. Outside CLASS tree, two buffers (input and output) are connected to the root processor as the storage of the messages transmitted between the host computer and the tree machine. It is easy to describe the tree structure by a diagram as in Fig.6-1.

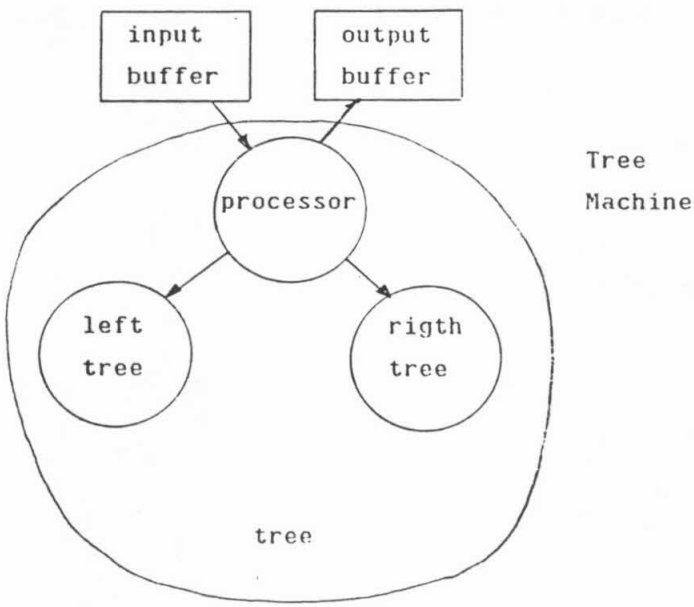


Fig.6-1 The Structure of CLASS tree

6.1 The Structure of the Simulator

Fig.6-2 shows the data structure of the simulator. Class treemachine contains the whole structure in Fig.6-1. A random access disk file fvm

simulates the virtual memory of the tree machine. It is provided to prevent running out of memory space of the simulator. It doesn't exist in the real tree machine. Class processor simulates the architecture of the tree machine processor. The real tree machine processor has 4K nibble memory. In class processor, only a half K nibble memory is given. The 4K memory is divided into 32 pages. One page is 128 nibbles in length. The memory in the simulator can only contain 4 pages of memory nibbles at the same time. A page table keeps the page numbers currently stored in the simulator memory. The file fvm stores the memory contents of all the processors in the tree in the sequence of the procno of the processors. When a page fault occurs (the page number is not shown in the page table), the first page in the page table is stored into the corresponding location of the fvm file and the new page is copied into the memory location of the deleted page from the fvm file. The new page number is then appended to the end of the page table. Class tree is a recursive tree structure. We can create a binary tree with arbitrary size under class tree.

```
CLASS treemachine;
BEGIN
   !*************************************************;
   ! Input/Output Buffers, interface to the host computer;
   !*************************************************;
    REF(register)inbuf,outbuf;     !16 bits I/O buffers;
    BOOLEAN fginbuf,fgoutbuf;      !flags of I/O buffers;
    REF(tree)tr;                   !tree machine;
    REF(directfile)fvm;            !virtual memory file;

    inbuf:-NEW register(16);
    outbuf:-NEW register(16);
    fginbuf:=true; fgoutbuf:=false;
    tr:-new tree;
END of CLASS tree machine;

!Tree Machine Processor;
CLASS processor;
BEGIN
   !*************************************************;
   ! The Hardware Components in the TMP ;
   !*************************************************;
    REF(register) ARRAY R[0:15];   !16 12-bit general purpose registers;
    REF(register) ARRAY PI[0:3];   !16-bit Input fifo of four I/O ports;
    REF(register) ARRAY PO[0:3];   !16-bit Output fifo of four I/O ports;
    BOOLEAN ARRAY FI[0:3];  !Flag to indicate the empty of input fifo;
    BOOLEAN ARRAY FO[0:3]; !Flag to indicate the full of output fifo;
    BOOLEAN Z,N,C,V;        !Four flags;
    REF(register)IR;        !4-bit instruction register;
    REF(register)PC;        !12-bit program counter;
    INTEGER ARRAY MEM[0:511];     !Memory;

   !*************************************************;
   ! Virtual Momory Page Table;
   !*************************************************;
    INTEGER procNo;                 !The processor number;
    INTEGER ARRAY pageTbl[1:2,1:4]; !4 entry page table;
END of CLASS processor;

!A recursive tree structure;
CLASS tree;
BEGIN
    REF(processor)pros;       !the root processor;
    REF(tree)lb,rb;           !the left and right subtrees;
    pros:-NEW processor;
END of CLASS tree;
```

Fig.6.2 The Data Strucure of the Simulator

6.2 The Interface Commands of the Simulator

The simulator runs interactively. After execution starts, the simulator cleans the screen and promts a 'TS>' on the screen. The user can type one of the following commands or their nonambiguous abbreviations. The simulator accepts the command, does the proper work and promts the proper messages. After the work is done, another 'TS>' will be promted out and the simulator is ready to accept a new command. The list of the interface commands and the syntax is shown as follows:

Manipulation of the tree structure

CREATE <integer> -- Create an <integer>-level tree structure

KILL -- Kill the tree structure

Loading and Simulation

LOAD <fileName> -- Run the downloading program decribed in Chapter 5 and load the program code from <fileName>.TBJ into the tree machine. A CREATE command should be given before a LOAD command to create the tree machine with the proper height. After downloading is finished, a message "Start Execution, waiting for input...." will be prompted out on the screen. Then, the user can issue an INPUT command to start running the simulator.

INPUT <msgName>, arg1,arg2,.... -- <msgName> can be an integer or an identifier. Users can get the valid message name & number pairs by a SHOW command. Any number of arguments can be appended after <msgName>. The simulator puts the arguments into a queue and takes the message number and the first argument into the input buffer of the tree machine. If the input buffer is already full, then this input command will not be accepted and the simulator will prompt a warning message.

RUN -- Runs one simulation step. One simulator step means a sequence of steps: 1). If the input buffer is full and the external

input port of the root is empty, then the content of the input buffer is taken into the external input port of the root processor and flags are set. 2). All the nodes in the tree execute one instruction. 3). Transmit all the messages in the input or output ports of the nodes to the destination ports and set the flags. 4). If the external output port of the root is full, then the contents of the port is placed into the output buffer of the tree machine and the flags are set. 5). If the output buffer is full, then the message name (looked up from the .TBL file) and its arguments is prompted on the screen.

PROCEDE -- Run the simulator until the root processor returns to the wait mode. Update the system clock and prompt it when the execution is finished.

RESET -- Reset the system clock and the program counters of all the nodes

Informations

DUMP -- Dump out all the memory contents in the tree machine

SHOW -- Show the map of the message & number pairs of the root processor

Others

HELP -- Get this help file

EXIT -- Terminate the simulation

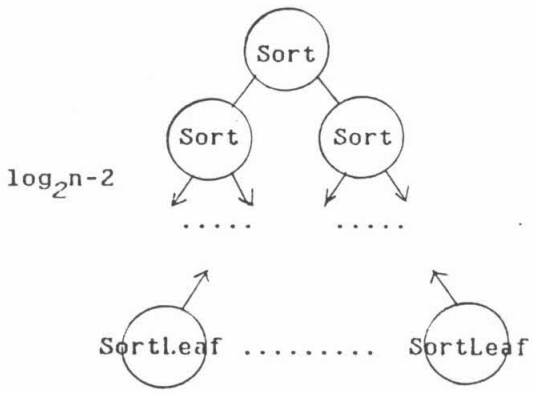
VII. The Simulation Result for Some Problems

Three of Sally's algorithms [1] have been selected to test the tree machine's performance. They are the tree sorting algorithm, the largest clique algorithm and the two by two matrix multiplication algorithm. One similarity between these three algorithms is that all of them are designed on a binary tree. One additional test program has been designed to test the difference of the binary tree and the n-ary tree.

7.1 The Sorting Problem

The sorting problem is a typical problem suitable for a tree structure. Sally's algorithm is defined on a balanced, binary tree with height $\log_2 n$, where n is the total number of elements that will be sorted. Each processor in the tree keeps one element itself. The largest number is always kept at the top of the tree. The procedure load will load one element down to the tree. The procedure unload will release the largest element in the tree and rearrange the data so that the second largest element will appear at the root of the tree. The space complexity of this algorithm is O(n) and the predicted time complexity is O(n) too.

The connection plan of the tree is shown in Fig.7-1. There are two different modules in the tree, i.e. sort and sortleaf. A 4-level sorting tree, which can sort 15 elements, has been tested and some figures are shown in Table 7.1.




```
CONNECT
  ROOT     SORT(2)
  DO       M
   NODE    SORT(2)
  EDO
  NODE     SORTLEAF
ENDC

  M = logn-2,
```

Fig.7-1 The Sorting Tree

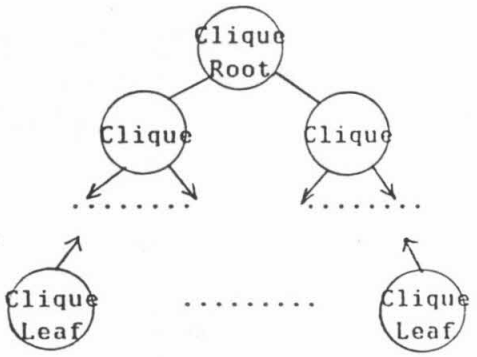
7.2 The Largest Clique Problem

To find the largest clique in an undirected graph is an NP-complete problem. It takes exponential time to solve the largest clique problem in a sequential machine. By Sally's algorithm, it takes only O(n) time with a (n+1)-level binary tree. So the total number of processors in the tree is $2^{n+1}-1$. And the time-space performance is in the order of $n \times 2^{n+1}$.

The clique tree is built so that each path from the root to one leaf node represents a possible clique of the graph. The algorithm will check all of the 2^n possible cliques simultaneously. It can be done in linear

time because the height of the tree is n+1.

There are three different modules in the clique tree, cliqueroot, clique and cliqueleaf. The connection plan is shown in Fig.7-2. A 3 node, 2 arc graph and a 4 node, 6 arc graph have been tested and the data is shown in Table 7.1.




```
CONNECT
  ROOT     CLIQUEROOT(2)
  DO       N-1
   NODE    CLIQUE(2)
  EDO
  NODE     CLIQUELEAF
ENDC
```

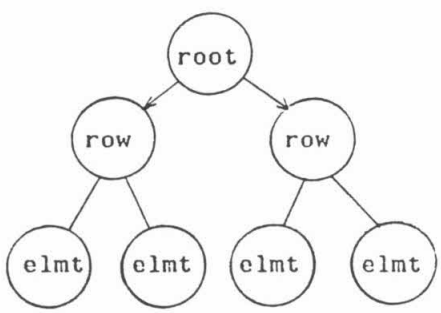
Fig.7-2 The Largest Clique Tree

7.3 The Matrix Multiplication Problem

Sally's matrix multiplication algorithm is defined on a three level, n-ary tree. In general, a n x m matrix multiplies a m x k matrix to get a n x k matrix product. The root of the tree has fanout n (the column of the multiplicand) and the nodes in the second level of the tree have fanout m (the row of the multiplicand). There are three different modules in the multiplication tree, root, row and element. The root gets the elements of the multiplicand row by row (in the sequence of $a_{11}, a_{12}, \ldots, a_{1m}, a_{21}, \ldots, a_{2m}, \ldots, a_{n1}, \ldots a_{nm}$) and sends the elements of the first row down to its first child and so on. The row processor gets one row of the multiplicand and sends the first element of the row to its first child and so on. Hence, the elements in the matrix are all stored at the bottom of the tree. One element processor keeps one element. After the multiplicand has been loaded into the tree, the root takes one column of the multiplier at a time, and sends them to all its children. The row processor distributes the elements to its children from left to right. The element processor will multiply the column element of the multiplier with the row element of the multiplicand it kept and send the product back to its father. Then, the row element receives all the products from its children and adds them up to get one element of the product matrix.

The total number of processors in the multiplication tree is nxm+m+1. For a n by n matrix, the space complexity is $O(n^2)$. The time complexity is in the order of n^2 because the data elements should be loaded into or unloaded from the tree individually. Thus, the time-space complexity of the tree is $O(n^4)$ compared to $O(n^3)$ for the sequential machine.

In the previous discussion, we just considered the logical tree. Nevertheless, when a nonbinary logical tree is converted to a binary tree, the additional padding processors will effect the space performance of that tree in order of magnitude. The extra codes for the padding processors and the father processors will increase the downloading time and the longer communication path will increase the execution time, so that it will effect the time performance too. So, we have to reevaluate the performance of the matrix multiplication algorithm by consideration of the padding processors. At this moment, I have only tested the program on a 2 by 2 matrix to avoid padding processors. The padding problem will be discussed in Section 7.4. The connection plan of a 2 x 2 matrix multiplication tree is shown in Fig.7-3.




```
CONNECT
  ROOT    ROOT(2)
  NODE    ROW(2)
  NODE    ELEMENT
ENDC
```

Fig.7-3 The 2x2 Matrix Multiplication Tree

Table 7.1 is a summary of the simulation results of these three algorithms. We notice that the downloading time is much longer than the execution time. The downloading time is determined by the length of the total program code plus the total number of nodes in the tree. An approximated equation has been shown in Section 5.3.

Table 7.1
The Comparison of the Tree Algorithms

	Largest 3 nodes	Cilque 4 nodes	Tree Sort 15 points	Matrix Mult 2 x 2
level	4	5	4	3
nodes	15	31	15	7
# dif. modules	3	3	2	3
total codes !	782	782	470	462
length of loader !	312	312	312	312
downloading time	2444	2624	1553	1420
execution time	170	269	1157	273
	(2 arcs)	(6 arcs)	(14 numbers)	

** Downloading time = 3 x (# of program codes + # of nodes x 3)
! number of nibbles

7.4 Binary Tree vs N-ary Tree

A very simple test program has been written and tested under the tree software system. This program is used to test the performance of trees with different fanout. In table 7.2, we can see how the padding algorithm affects the loading time and the execution time and the program size of the tree machine.

Because of the insertion of the padding code, the program size blows up tremendously. In the same way, the downloading time is increasing badly. In this specific example, the program size and the downloading time increase about 2.5 times. The code for one padding processor is about 330 nibbles and the extra code while converting a pair of simple input/output instructions to I/O macros in the father node is about 130 nibbles. The total increase of the program code depends on how many different padding processors should be inserted. If there are n different padding processors, then the extra code is about n x (330+130) = 460 x n nibbles. It is about the size of one tree sort program or one matrix multiplication

increses because of the longer communication path and extra arguments and operations for each message transmission. Compared to the increase of the downloading time, the increase of the execution time is not significant.

Table 7.2
The Comparison of Different Fanout Trees

	Fanout 2	Fanout 3	Fanout 5
level	2	3	4
total nodes	7	15	31
nodes in use	7	11	19
padding nodes	0	1	3
# diff. modules	3	4	4
total code	283	748	748
padding code	0	335	335
codes of father node	97	227	227
downloading time	905	2379	2379
execution time	42	65	65

VIII. Conclusion and Further Plan

From the results in Chapter 7, some observations have been made:

1. From Table 7.1 and Table 7.2, we can see that the downloading time is longer than the execution time by at least an order of magnitude. Even worse, the downloading time will increase exponentially with the increase of the tree depth. When we consider the time complexity of an algorithm, we always neglect the loading time. Is it a true and accurate estimation for any tree algorithm? I believe the answer should be 'NO'. A tree structure can provide some extent of concurrency on computation. But the communication of the tree is a kind of sequential operation intrinsically. All the messages will be sent to the tree through the root processor. The time performance of many tree algorithms is restricted by the number of

messages passing through the root. If we consider the time to pass all the program code to the tree sequentially, the loading time will be a dominant factor for the performance of a tree algorithm.

2. From the result shown in Table 7.2, the following three problems are important. 1). Our tree model allows a user to describe arbitrary fanout trees to make the tree model more flexible and powerful. But it turns out that we lose the speed advantange on the transfer to a binary tree. Although we make the tree model more attractive but the tree machine seems to be more unattractive. Should more restrictions be placed on the specification so that we can shorten the gap between a logical tree model and the physical tree machine? 2). The padding algorithms I proposed is certainly not the best. How to improve the efficiency of the padding algorithms (in other word, the size of the padding code) is worth to investigate. 3). Is there another way to solve the mapping problem other than inserting the padding processors? For a strictly binary tree, it seems hard to find another method to do the mapping. But if we implement the tree on a symmetric geometric structure such as a matrix, a hexagonal structure, maybe we can find a good mapping algorithm without inserting additional processors.

3. One important problem that I haven't discussed in this report is that using a physical tree of smaller size than the logical tree. The physical tree has fixed size but the logical model can be unlimited large. Once the physical tree is not large enough, we have to remap the logical tree and time-share each processor in the physical tree. The mapping algorithm should prevent congestion and deadlock situations and optimize the communication length. Since the memory size of a processor is limited, it is better to put two or more nodes with the same program in one physical processor to save memory. A job scheduler is necessary to handle the time-sharing situation.

All the questions I mentioned before are still open. If they are solvable, then the tree machine will be a very promising multiprocessor system. If they are not, then we have to reevaluate the performance of the tree machine.

REFERENCE

1. S. Browning, 'The Tree Machine: A Highly Concurrent Computing Environment', Ph.D. Thesis, Dept. of Computer Science, Caltech, 1980

2. Caltech TMP Group, 'The Instruction Set of the TMP', unpublished internal document, Caltech, 1980

3. S. Browning, 'Generating Padding Processors for Arbitrary Fanout Trees', display file #3827, Dept. of Computer Science, Caltech, 1980

4. M. Chen & C. Mead, 'HARMOS-A Notation for Designing Concurrent Systems', display file #3927, Dept. of Computer Science, Caltech, 1980

APPENDIX A The Instruction Set of the TMP

Memory Manipulation

LOAD@ Rs Rd

STORE@ Rd Rs

Unary Operations

Unop R subop

		Flags Affected
TST	(R) ==> (R)	Z N - -
CLR	0 ==> (R)	- - - -
ONE	1 ==> (R)	- - - -
SET	-1 ==> (R)	- - - -
INC	(R)+1 ==> (R)	Z N - V
DEC	(R)-1 ==> (R)	Z N - V
NEG	-(R) ==> (R)	Z N C V
NEGC	-(R)-1+C ==> (R)	Z N C V
COM	~(R) ==> (R)	Z N - -
ASR	MSB(R)□(R) ==> (R)□C	Z N C V
LSR	0□(R) ==> (R)□C	Z N C V
ASL	(R)□0 ==> C□(R)	Z N C V
ROR	C□(R) ==> (R)□C	Z N C V
ROL	(R)□C ==> C□(R)	Z N C V

Binary Operations

Three forms: The first one is register-register operations. The other two forms are register-immediate value operations.

Binop Rs Rd op S=(Rs)

Binsi Rd I op S=[0,0,I]

```
Binli   Rd    I2    I1    I0    op            S=[I2,I1,I0]
```

		Flags Affected
MOV	S ==> (Rd)	Z N - -
AND	(Rd) and S ==> (Rd)	Z N - -
OR	(Rd) or S ==> (Rd)	Z N - -
XOR	(Rd) xor S ==> (Rd)	Z N - -
ADD	(Rd) + S ==> (Rd)	Z N C V
ADDC	(Rd) + S + C ==> (Rd)	Z N C V
SUB	(Rd) - S ==> (Rd)	Z N C V
SUBC	(Rd)-S-1+C ==> (Rd)	Z N C V
CMP	(Rd) - S : Set flags only	Z N C V
BIT	(Rd) and S : Set flags only	Z N - -

Control of flow

```
Jmp@    R

Call    R     A2    A1    A0
```

Branch Instructions

```
Brat    D1    D0    cond        D=[D1,D2]

Braf    D1    D0    cond        D=[D1,D2]
```

16 branch conditions:

BI0	Port 0 input register empty (FI0=1)
BO0	Port 0 output register full (FO0=1)
BI1	Port 1 input register empty (FI1=1)
BO1	Port 1 output register full (FO1=1)
BI2	Port 2 input register empty (FI2=1)
BO2	Port 2 output register full (FO2=1)
BI3	Port 3 input register empty (FI3=1)
BO3	Port 3 output register full (FO3=1)
BE	equal (Z=1)
BLE	less than or equal (N xor V or Z=1)
BLT	less than (N xor V=1)
BULE	unsigned less than or equal (not C or Z=1)
BUL	unsigned less than (C=0)

```
BOV     overflow (V=1)
BN      minus (N=1)
BR      always
```

Input/Output Instructions

```
INW     00.1.0      msg

In      PT.1.1      R       msg

Out     PT.1.x      R       msg
```

Flag Manipulation

```
SetF      Z.N.C.V

StoreF      R
```

APPENDIX B The Syntax of the Assembly Language

Format:

```
+---------------------------------------+
| label    | mnemonic   operands        |
+---------------------------------------+
 1       10 11                        80
```

lable is a 1 to 10 alphanumeric character string.
mnemonic is one of the following instructions starting at location 11.
operands are separated with mnemonic by at least one blank character.
The upper case letter is the same as the lower case letter.
Comment line starting with '!' can be inserted anywhere in the program.

Instructions:

MNEMONIC INSTRUCTIONS

```
LOAD    Rs, Ad
STORE   Rd, Ad
```

! Rs or Rd is one of the 16 registers, R1,R2,.....,R15.
! Ad can be an identifier or a register. If it is a register, then, that register stores the memory address.

Unary Operations

```
TST     R                ! R is one of the 16 registers.
CLR     R
ONE     R
SET     R
INC     R
DEC     R
NEG     R
NEGC    R
COM     R
ASR     R
LSR     R
ASL     R
```

```
ROR     R
ROL     R
```

Binary Operations

```
MOV     Rs, Rd          ! Rs is either a register or an integer.
AND     Rs, Rd          ! Rd is a register
OR      Rs, Rd
XOR     Rs, Rd
ADD     Rs, Rd
ADDC    Rs, Rd
SUB     Rs, Rd
SUBC    Rs, Rd
CMP     Rs, Rd
BIT     Rs, Rd

JMP     R
CALL    AD, R
```

Conditional Branch Operations

Branch on True

```
BI1,T   Ad              ! Ad is an expression, which has the form
BO1,T   Ad                <id1>[*<id2>][(+|-)<id2>]
BI2,T   Ad                <id1> can be a identifier, an integer or '*'
BO2,T   Ad                (the value of the current locaton counter).
BI3,T   Ad                <id2> can be a identifier or an integer
BO3,T   Ad
BI4,T   Ad
BO4,T   Ad
BEQ,T   Ad
BLE,T   Ad
BLT,T   Ad
BULE,T  Ad
BUL,T   Ad
BOV,T   Ad
BM,T    Ad
BR      Ad              ! Branch Always
```

Branch on False

```
BI1,F    Ad          ! The input register of Port 1 is not empty
BO1,F    Ad          ! The output register of Port 1 is not full
BI2,F    Ad          ! The input register of Port 2 is not empty
BO2,F    Ad          ! The output register of Port 2 is not full
BI3,F    Ad
BO3,F    Ad
BI4,F    Ad
BO4,F    Ad
BEQ,F    Ad          ! Branch on not equal
BLE,F    Ad          ! Branch on not less than or equal
BLT,F    Ad          ! Branch on not less than
BULE,F   Ad          ! Branch on not unsigned less than or equal
BUL,F    Ad          ! Branch on not unsigned less than
BOV,F    Ad          ! Branch on not overflow
BM,F     Ad          ! Branch on not minus
```

Input/Output Operations

```
INW      msg#                ! Dispatch input, only connected to port 1
IN       port,msg#,(arg1,arg2,...)   ! Input macro
OUT      port,msg#,(arg1,arg2,...)   ! Output macro
```

! port can be integer 1 to 4, or a logical name which has been defined in EXT or INT pseudo commands.

! msg# can be a integer 1 to 16, or a logical name.

! The argument list IN or OUT macros may have none to arbitrary number of argments. The arguments can be an identifier, a register or a integer (in IN macro only). The arguments are separated by comma, If there is only one argument, the parentheses can be omitted.

PSEUDO INSTRUCTIONS

Connection Plan Definition

```
CONNECT
ROOT     name(fanout)
NODE     [integer]name[(fanout)] {,[integer]name[(fanout)]} |
```

```
          name(lower:upper)[(fanout)]
SUBTREE   name
ENDS
ENDC
```

Module, procedure start and end

```
PROGRAM   name
END
MODULE    name
ENDM
PROC      name
ENDP
```

Port Declaration

```
PORT
EXT       name
INT       name{,name} | name(lower:upper)
```

Memory Allocation, Variable Initialization

```
label   BWS   expression      !Block word storage (3 nibbles per word)
label   DWF   expression      !Define word field (set initial value)
label   EQU   expression      !Define Constant
```

Program Control Instructions

```
DO        expression          !Do loop
EDO
```

Macro Pseudo

```
MACRO                         !Macro Start
EMC                           !Macro end
```

!All the square brackets and braces in the syntax are metasymbols.